

8

Coding and Design Patterns

Now that you know about the object-oriented features of JavaScript, such as prototypes and inheritance, and you have seen some practical examples of using the browser objects, let's move forward, or rather, move a level up. Let us have a look at some common patterns of JavaScript utilization. First, let's define what a pattern is. In short, a pattern is a good solution to a common problem.

Sometimes when you are facing a new programming problem, you might recognize right away that you've previously solved another, suspiciously similar problem. In such cases, it is worth isolating this class of problems and searching for a common solution. A pattern is a proven and reusable solution (or an approach to a solution) to a class of problems. Sometimes a pattern is nothing more than an idea or a name, but sometimes just using a name helps you think more clearly about a problem. Also, when working with other developers in a team, it's much easier to communicate when everybody uses the same terminology when discussing a problem or a solution.

Sometimes there might be cases when your problem is rather unique and doesn't fit into a known pattern. Blindly applying a pattern just for the sake of using a pattern is not a good idea. It's actually better to not use a pattern (if you can't come up with a new one) than to try and change your problem so that it fits an existing solution.

This chapter talks about two types of patterns:

- Coding patterns – these are mostly JavaScript-specific best practices
- Design patterns – these are language-independent patterns, popularized by the "Gang of Four" book

Coding Patterns

This first part of the chapter discusses some patterns that reflect JavaScript's unique features. Some patterns aim to help you with organizing your code (such as namespace patterns), others are related to improving performance (such as lazy definitions and init-time branching), and some make up for missing features such as privately scoped properties. The patterns discussed in this section include:

- Separating behavior
- Namespaces
- Init-time branching
- Lazy definition
- Configuration objects
- Private variables and methods
- Privileged methods
- Private functions as public methods
- Self-executable functions
- Chaining
- JSON

Separating Behavior

As discussed previously, the three building blocks of a web page are:

- Content (HTML)
- Presentation (CSS)
- Behavior (JavaScript)

Content

HTML is the content of the web page; the actual text. The content should be marked up using the smallest amount of HTML tags that sufficiently describe the semantic meaning of that content. For example, if you're working on a navigation menu it's probably a good idea to use `` and `` as a navigation menu is basically a list of links.

Your content (HTML) should be free from any formatting elements. Visual formatting belongs to the presentation layer and should be achieved through the use of CSS (Cascading Style Sheets). This means that:

- The `style` attribute of HTML tags should not be used, if possible.
- Presentational HTML tags such as `` should not be used at all.
- Tags should be used for their semantic meaning, not because of how browsers render them by default. For instance, developers sometimes use a `<div>` tag where a `<p>` would be more appropriate. It's also favorable to use `` and `` instead of `` and `<i>` as the latter describe the visual presentation rather than the meaning.

Presentation

A good approach to keep presentation out of the content is to reset, or nullify, all browser defaults; for example using `reset.css` from the Yahoo! UI library. This way the browser's default rendering won't distract you from consciously thinking about the proper semantic tags to use.

Behavior

The third component of a page is the behavior. Behavior should be kept separate from both the content and the presentation. Behavior is usually added by using JavaScript that is isolated to `<script>` tags, and preferably contained in external files. This means not using any inline attributes such as `onclick`, `onmouseover`, and so on. Instead of that, you can use the `addEventListener/attachEvent` methods that you have already seen in the previous chapter.

The best strategy for separating behavior from content would be:

- Minimize the number of `<script>` tags
- Avoid inline event handlers
- Do not use CSS expressions
- Dynamically add markup that has no purpose when JavaScript is disabled by the user
- Towards the end of your content, when you are ready to close the `<body>` tag, insert a single external `.js` file

Example of Separating Behavior

Let's say you have a search form on a page and you want to validate the form with JavaScript. So you go ahead and keep the form tags free from any JavaScript and then, immediately before the closing `</body>` tag, you insert a `<script>` tag which links to an external file.

```
<body>
  <form id="myform" method="post" action="server.php">
    <fieldset>
      <legend>Search</legend>
      <input
        name="search"
        id="search"
        type="text"
      />
      <input type="submit" />
    </fieldset>
  </form>
  <script type="text/javascript" src="behaviors.js"></script>
</body>
```

In `behaviors.js` you attach an event listener to the submit event. In your listener, you check to see if the text input field was left blank and if so, stop the form from being submitted. Here's the complete content of `behaviors.js`. It assumes that you've created your `myevent` utility from the exercise at the end of the previous chapter:

```
// init
myevent.addListener('myform', 'submit', function(e){
  // no need to propagate further
  e = myevent.getEvent(e);
  myevent.stopPropagation(e);
  // validate
  var el = document.getElementById('search');
  if (!el.value) { // too bad, field is empty
    myevent.preventDefault(e); // prevent the form submission
    alert('Please enter a search string');
  }
});
```

Namespaces

Global variables should be avoided in order to lower the possibility of variable naming collisions. One way to minimize the number of globals is by namespacing your variables and functions. The idea is simple: you create only one global object and all your other variables and functions become properties of that object.

An Object as a Namespace

Let's create a global object called `MYAPP`:

```
// global namespace
var MYAPP = MYAPP || {};
```

Now instead of having a global `myevent` utility (from the previous chapter), you can have it as an event property of the `MYAPP` object.

```
// sub-object
MYAPP.event = {};
```

Adding the methods to the event utility is pretty much the same as usual:

```
// object together with the method declarations
MYAPP.event = {
  addListener: function(el, type, fn) {
    // .. do the thing
  },
  removeListener: function(el, type, fn) {
    // ...
  },
  getEvent: function(e) {
    // ...
  }
  // ... other methods or properties
};
```

Namespaced Constructors

Using a namespace doesn't prevent you from creating constructor functions. Here is how you can have a DOM utility that has an `Element` constructor which allows you to create DOM elements more easily.

```
MYAPP.dom = {};
MYAPP.dom.Element = function(type, prop){
  var tmp = document.createElement(type);
  for (var i in prop) {
    tmp.setAttribute(i, prop[i]);
  }
  return tmp;
}
```

Similarly, you can have a `Text` constructor to create text nodes if you want to:

```
MYAPP.dom.Text = function(txt){
  return document.createTextNode(txt);
}
```

Using the constructors to create a link at the bottom of a page:

```
var e11 = new MYAPP.dom.Element(
    'a',
    {href: 'http://phpied.com'}
);
var e12 = new MYAPP.dom.Text('click me');
e11.appendChild(e12);
document.body.appendChild(e11);
```

A namespace() Method

Some libraries, such as YUI, implement a namespace utility method that makes your life easier, so that you can do something like:

```
MYAPP.namespace('dom.style');
```

instead of the more verbose:

```
MYAPP.dom = {};
MYAPP.dom.style = {};
```

Here's how you can create such a namespace() method. First you create an array by splitting the input string using the period (.) as a separator. Then, for every element in the new array, you add a property to your global object if such a property doesn't already exist.

```
var MYAPP = {};
MYAPP.namespace = function(name){
    var parts = name.split('.');
    var current = MYAPP;
    for (var i in parts) {
        if (!current[parts[i]]) {
            current[parts[i]] = {};
        }
        current = current[parts[i]];
    }
}
```

Testing the new method:

```
MYAPP.namespace('event');
MYAPP.namespace('dom.style');
```

The result of the above is the same as if you did:

```
var MYAPP = {
  event: {},
  dom: {
    style: {}
  }
}
```

Init-Time Branching

In the previous chapter, you saw that different browsers often have different implementations for the same or similar functionalities. In such cases, you need to branch your code depending on what's supported by the browser currently executing your script. Depending on your program this branching can happen far too often and, as a result, can slow down the script execution.

You can mitigate this problem by branching some parts of the code during initialization, when the script loads, rather than during runtime. Building upon the ability to define functions dynamically, you can branch and define the same function with a different body depending on the browser. Let's see how.

First, let's define a namespace and placeholder method for the event utility.

```
var MYAPP = {};
MYAPP.event = {
  addListener: null,
  removeListener: null
};
```

At this point, the methods to add or remove a listener are not implemented. Based on the results from feature sniffing, these methods can be defined differently.

```
if (typeof window.addEventListener === 'function') {
  MYAPP.event.addListener = function(el, type, fn) {
    el.addEventListener(type, fn, false);
  };
  MYAPP.event.removeListener = function(el, type, fn) {
    el.removeEventListener(type, fn, false);
  };
} else if (typeof document.attachEvent === 'function'){ // IE
  MYAPP.event.addListener = function(el, type, fn) {
    el.attachEvent('on' + type, fn);
  };
}
```

```
    MYAPP.event.removeListener = function(el, type, fn) {
        el.detachEvent('on' + type, fn);
    };
} else { // older browsers
    MYAPP.event.addListener = function(el, type, fn) {
        el['on' + type] = fn;
    };
    MYAPP.event.removeListener = function(el, type, fn) {
        el['on' + type] = null;
    };
};
```

After this script executes, you have the `addListener()` and `removeListener()` methods defined in a browser-dependent way. Now every time you invoke one of these methods it will not do any more feature sniffing, and as a result will run faster because it is doing less work.

One thing to watch out for when sniffing features is not to assume too much after checking for one feature. In the example above, this rule is broken because the code only checks for `add*` support but then defines both the `add*` and the `remove*` methods. In this case it's probably safe to assume that in a next version of the browser, if IE decides to implement `addEventListener()` it will also implement `removeEventListener()`. But imagine what happens if IE implements `stopPropagation()` but not `preventDefault()` and you haven't checked for these individually. You have assumed that because `addEventListener()` is not defined, the browser is IE and write your code using your knowledge of how IE works. Remember that all of your knowledge is based on the way IE works today, but not necessarily the way it will work tomorrow. So to avoid many rewrites of your code as new browser versions are shipped, it's best to individually check for features you intend to use and don't generalize on what a certain browser supports.

Lazy Definition

The *lazy definition* pattern is very similar to the previous *init-time branching* pattern. The difference is that the branching happens only when the function is called for the first time. When the function is called, it redefines itself with the best implementation. Unlike the *init-time branching* where the `if` happens once, during loading, here it might not happen at all – in cases when the function is never called. The lazy definition also makes the initialization process lighter, as there's no *init-time branching* work to be done.

Let's see an example that illustrates this, via the definition of an `addListener()` function. The function is first defined with a generic body. It checks which functionality is supported by the browser when it is called for the first time and then redefines itself using the most suitable implementation. At the end of the first call, the function calls itself so that the actual event attaching is performed. The next time you call the same function it will be defined with its new body and will be ready for use, so no further branching is necessary.

```
var MYAPP = {};  
MYAPP.myevent = {  
  addListener: function(el, type, fn){  
    if (typeof el.addEventListener === 'function') {  
      MYAPP.myevent.addListener = function(el, type, fn) {  
        el.addEventListener(type, fn, false);  
      };  
    } else if (typeof el.attachEvent === 'function'){  
      MYAPP.myevent.addListener = function(el, type, fn) {  
        el.attachEvent('on' + type, fn);  
      };  
    } else {  
      MYAPP.myevent.addListener = function(el, type, fn) {  
        el['on' + type] = fn;  
      };  
    }  
    MYAPP.myevent.addListener(el, type, fn);  
  }  
};
```

Configuration Object

This pattern is useful when you have a function or method that accepts a lot of parameters. It's up to you to decide how many constitutes "a lot", but generally a function with more than three parameters is probably not very convenient to call, as you have to remember the order of the parameters, and is even more inconvenient when some of the parameters are optional.

Instead of having many parameters, you can use one parameter and make it an object. The properties of the object are the actual parameters. This is especially suitable for passing configuration parameters, because these tend to be numerous and mostly optional (with smart defaults). The beauty of using a single object as opposed to multiple parameters is:

- The order doesn't matter
- You can easily skip parameters that you don't want to set

- It makes the function signature easily extendable should future requirements necessitate this
- It makes the code more readable because the config object's properties are present in the calling code, along with their names

Imagine you have a `Button` constructor used to create input buttons. It accepts the text to put inside the button (the `value` attribute of the `<input>` tag) and an optional parameter of the `type` of button.

```
// a constructor that creates buttons
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.Button = function(text, type) {
    var b = document.createElement('input');
    b.type = type || 'submit';
    b.value = text;
    return b;
}
```

Using the constructor is simple; you just give it a string. Then you can append the new button to the body of the document:

```
document.body.appendChild(new MYAPP.dom.Button('puuush'));
```

This is all well and works fine, but then you decide you also want to be able to set some of the style properties of the button, such as colors and fonts. You might end up with a definition like:

```
MYAPP.dom.Button = function(text, type, color, border, font) {
    // ....
}
```

Now using the constructor can become a little inconvenient, for example when you want to set the third and fifth parameter, but not the second or the fourth:

```
new MYAPP.dom.Button('puuush', null, 'white', null,
    'Arial, Verdana, sans-serif');
```

A better approach is to use one config object parameter for all the settings. The function definition can become something like:

```
MYAPP.dom.Button = function(text, conf) {
    var type = conf.type || 'submit';
    var font = conf.font || 'Verdana';
    // ...
}
```

Using the constructor:

```
var config = {
  font: 'Arial, Verdana, sans-serif',
  color: 'white'
};
new MYAPP.dom.Button('puuush', config);
```

Another usage example:

```
document.body.appendChild(
  new MYAPP.dom.Button('dude', {color: 'red'})
);
```

As you can see, it's easy to set only selected parameters and to switch around their order. In addition, it's friendlier and makes the code easier to understand when you see the names of the parameters when you call the method.

Private Properties and Methods

JavaScript doesn't have the notion of *access modifiers*, which set the privileges of the properties in an object. Classical languages often have access modifiers such as:

- Public – all users of an object can access these properties (or methods)
- Private – only the object itself can access these properties
- Protected – only objects inheriting the object in question can access these properties

JavaScript doesn't have a special syntax to denote private properties but, as discussed in Chapter 3, you can use local variables and methods inside a constructor and achieve the same level of protection.

Continuing with the example of the `Button` constructor, you can have a local variable `styles` which contains all the defaults, and a local `setStyle()` function. These are invisible to the code outside of the constructor. Here's how `Button` can make use of the local private properties:

```
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.Button = function(text, conf) {
  var styles = {
    font: 'Verdana',
    border: '1px solid black',
    color: 'black',
    background: 'grey'
```

```
};
function setStyles() {
    for (var i in styles) {
        b.style[i] = conf[i] || styles[i];
    }
}
conf = conf || {};
var b = document.createElement('input');
b.type = conf['type'] || 'submit';
b.value = text;
setStyles();
return b;
};
```

In this implementation, `styles` is a private property and `setStyle()` is a private method. The constructor uses them internally (and they can access anything inside the constructor), but they are not available to code outside of the function.

Privileged Methods

Privileged methods (this term was coined by Douglas Crockford) are normal public methods that can access private methods or properties. They can act like a bridge in making some of the private functionality accessible but in a controlled manner, wrapped in a privileged method.

Continuing with the previous example, you can create a `getDefaults()` method that returns `styles`. In this way the code outside the `Button` constructor can see the default styles but cannot modify them. In this scenario `getDefaults()` would be a privileged method.

Private Functions as Public Methods

Let us say you've defined a function that you absolutely need to keep intact, so you make it private. But you also want to provide access to the same function so that outside code can also benefit from it. In this case, you can assign the private function to a publicly available property.

Let's define `_setStyle()` and `_getStyle()` as private functions, but then assign them to the public `setStyle()` and `getStyle()`:

```
var MYAPP = {};
MYAPP.dom = (function(){
    var _setStyle = function(el, prop, value) {
```

```
        console.log('setStyle');
    };
    var _getStyle = function(el, prop) {
        console.log('getStyle');
    };
    return {
        setStyle: _setStyle,
        getStyle: _getStyle,
        yetAnother: _setStyle
    };
})();
```

Now if you call `MYAPP.dom.setStyle()`, it will invoke the private `_setStyle()` function. You can also overwrite `setStyle()` from the outside:

```
MYAPP.dom.setStyle = function(){alert('b')};
```

Now the result will be:

- `MYAPP.dom.setStyle` points to the new function
- `MYAPP.dom.yetAnother` still points to `_setStyle()`
- `_setStyle()` is always available when any other internal code relies on it to be working as intended, regardless of the outside code

Self-Executing Functions

Another useful pattern that helps you keep the global namespace clean is to wrap your code in an anonymous function and execute that function immediately. This way any variables inside the function are local (as long as you use the `var` statement) and are destroyed when the function returns, if they aren't part of a closure. This pattern was discussed in more detail in Chapter 3.

```
(function(){
    // code goes here...
})();
```

This pattern is especially suitable for one-off initialization tasks performed when the script loads.

The self-executable pattern can be extended to create and return objects. If the creation of these objects is more complicated and involves some initialization work, then you can do this in the first part of the self-executable function and return a single object, which can access and benefit from any private properties in the top portion:

```
var MYAPP = {};  
MYAPP.dom = function(){  
  // initialization code...  
  function _private(){  
    // ... body  
  }  
  return {  
    getStyle: function(el, prop) {  
      console.log('getStyle');  
      _private();  
    },  
    setStyle: function(el, prop, value) {  
      console.log('setStyle');  
    }  
  };  
}();
```

Chaining

Chaining is a pattern that allows you to invoke methods on one line as if the methods are the links in a chain. This could be pretty convenient when calling several related methods. Basically, you invoke the next method on the result of the previous method, without the use of an intermediate variable.

Imagine you've created a constructor that helps you work with DOM elements. The code to create a new `` and add it to the `<body>` could look something like the following:

```
var obj = new MYAPP.dom.Element('span');  
obj.setText('hello');  
obj.setStyle('color', 'red');  
obj.setStyle('font', 'Verdana');  
document.body.appendChild(obj);
```

As you know, the constructors return the `this` object they create. You can make your methods such as `setText()` and `setStyle()` also return `this`, which will allow you to call the next method on the instance returned by the previous one. This way you can chain method calls:

```
var obj = new MYAPP.dom.Element('span');  
obj.setText('hello')
```

```
.setStyle('color', 'red')
.setStyle('font', 'Verdana');
document.body.appendChild(obj);
```

You might not even need the `obj` variable if you don't plan on using it after the new element has been added to the tree, so the code could look like:

```
document.body.appendChild(
    new MYAPP.dom.Element('span')
        .setText('hello')
        .setStyle('color', 'red')
        .setStyle('font', 'Verdana')
);
```

jQuery makes heavy use of the chaining pattern; this is probably one of the most recognizable features of this popular library.

JSON

Let's wrap up the coding patterns section of this chapter with a few words about JSON. JSON is not technically a coding pattern, but you can say that using JSON is a useful pattern.

JSON is a popular lightweight format for exchanging data. It's often preferred over XML when using `XMLHttpRequest()` to retrieve data from the server. JSON stands for JavaScript Object Notation and there's nothing specifically interesting about it other than the fact that it's extremely convenient. The JSON format consists of data, defined using object and array literals. Here is an example of a JSON string that your server could respond with after an XHR request.

```
{
  'name':    'Stoyan',
  'family': 'Stefanov',
  'books':  ['phpBB2', 'phpBB UG', 'PEAR']
}
```

An XML equivalent of this would be something like:

```
<?xml version="1.1" encoding="iso-8859-1"?>
<response>
  <name>Stoyan</name>
  <family>Stefanov</family>
  <books>
    <book>phpBB2</book>
    <book>phpBB UG</book>
    <book>PEAR</book>
  </books>
</response>
```

Firstly, you can see how JSON is lighter in terms of the number of bytes. But the main benefit is not the smaller byte size but the fact that it's extremely easy to work with JSON in JavaScript. Let's say you've made an XHR request and have received a JSON string in the `responseText` property of the XHR object. You can convert this string of data into a working JavaScript object by simply using `eval()`:

```
var obj = eval( '(' + xhr.responseText + ')' );
```

Now you can access the data in `obj` as object properties:

```
alert(obj.name); // Stoyan
alert(obj.books[2]); // PEAR
```

The problem with `eval()` is that it is insecure, so it's best if you use a little JavaScript library available from <http://json.org/> to parse the JSON data. Creating an object from a JSON string is still trivial:

```
var obj = JSON.parse(xhr.responseText);
```

Due to its simplicity, JSON has quickly become popular as a language-independent format for exchanging data and you can easily produce JSON on the server side using your preferred language. In PHP, for example, there are the functions `json_encode()` and `json_decode()` that let you serialize a PHP array or object into a JSON string, and vice versa.

Design Patterns

The second part of this chapter presents a JavaScript approach to a subset of the design patterns introduced by the book called *Design Patterns: Elements of Reusable Object-Oriented Software*, an influential book most commonly referred to as the *Book of Four* or the *Gang of Four*, or *GoF* (after its four authors). The patterns discussed in the *GoF* book are divided into three groups:

- *Creational patterns* that deal with how objects are *created* (instantiated)
- *Structural patterns* that describe how different objects can be *composed* in order to provide new functionality
- *Behavioral patterns* that describe ways for objects to *communicate* with each other

There are 23 patterns in the *Book of Four*, and more patterns have been identified since the book's publication. It is way beyond the scope of this book to discuss all of them, so the remainder of the chapter will demonstrate only four of them, along with examples of the implementation of these four in JavaScript. Remember that the patterns are more about interfaces and relationships rather than implementation. Once you have an understanding of a design pattern, it's often not difficult to implement it, especially in a dynamic language such as JavaScript.

The patterns discussed through the rest of the chapter are:

- Singleton
- Factory
- Decorator
- Observer

Singleton

Singleton is a *creational* design pattern meaning that its focus is on creating objects. It is useful when you want to make sure there is only one object of a given kind or class. In a classical language, this would mean that an instance of a class is only created once and any subsequent attempts to create new objects of the same class would return the original instance.

In JavaScript, because there are no classes, a singleton is the default and most natural pattern. Every object is a single object. If you don't copy it and don't use it as a prototype of another object, it will remain the only object of its kind.

The most basic implementation of the singleton in JavaScript is the object literal:

```
var single = {};
```

Singleton 2

If you want to use class-like syntax and still implement the singleton, things can become a bit more interesting. Let's say you have a constructor called `Logger()` and you want to be able to do something like:

```
var my_log = new Logger();
my_log.log('some event');
// ... 1000 lines of code later ...
var other_log = new Logger();
other_log.log('some new event');
alert(other_log === my_log); // true
```

The idea is that although you use `new`, only one instance needs to be created, and this instance is then returned in consecutive calls.

Global Variable

One approach would be to use a global variable to store the single instance. Your constructor could look like this:

```
function Logger() {
  if (typeof global_log === "undefined") {
    global_log = this;
  }
  return global_log;
}
```

Using this constructor gives the expected result:

```
var a = new Logger();
var b = new Logger();
alert(a === b); // true
```

The drawback is, of course, the use of a global variable. It can be overwritten at any time, even accidentally, and you lose the instance. The opposite—your global variable overwriting someone else's—is also possible.

Property of the Constructor

As you know, functions are objects and they have properties. You can assign the single instance to a property of the constructor function.

```
function Logger() {
  if (typeof Logger.single_instance === "undefined") {
    Logger.single_instance = this;
  }
  return Logger.single_instance;
}
```

If you write `var a = new Logger()`, `a` will point to the newly created `Logger.single_instance` property. A subsequent call `var b = new Logger()` will result in `b` pointing to the same `Logger.single_instance` property, which is exactly what you wanted.

This approach certainly solves the global namespace issue, because no global variables are created. The only drawback is that the property of the `Logger` constructor is publicly visible, so it can be overwritten at any time. In such cases, the single instance can be lost or modified.

In a Private Property

The solution to the problem of overwriting the publicly-visible property is to not use a public property, but a private one. You already know how to protect variables with a closure, so as an exercise you can implement this approach to the singleton pattern.

Factory

The factory is another creational design pattern as it deals with creating objects. The factory is useful when you have similar types of objects and you don't know in advance which one you want to use. Based on user input or other criteria, your code determines the type of object it needs on the fly.

Let's say you have three different constructors which implement similar functionality. The objects they create all take a URL but do different things with it. One creates a text DOM node; the second creates a link and the third, an image.

```
var MYAPP = {};  
MYAPP.dom = {};  
MYAPP.dom.Text = function() {  
  this.insert = function(when) {  
    var txt = document.createTextNode(this.url);  
    when.appendChild(txt);  
  };  
};  
MYAPP.dom.Link = function() {  
  this.insert = function(when) {  
    var link = document.createElement('a');  
    link.href = this.url;  
    link.appendChild(document.createTextNode(this.url));  
    when.appendChild(link);  
  };  
};  
MYAPP.dom.Image = function() {  
  this.insert = function(when) {  
    var im = document.createElement('img');  
    im.src = this.url;  
    when.appendChild(im);  
  };  
};
```

The way to use the three different constructors is exactly the same: you set the `url` property and then call the `insert()` method.

```
var o = new MYAPP.dom.Image();
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';
o.insert(document.body);
var o = new MYAPP.dom.Text();
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';
o.insert(document.body);
var o = new MYAPP.dom.Link();
o.url = 'http://images.packtpub.com/images/PacktLogoSmall.png';
o.insert(document.body);
```

Imagine your program doesn't know in advance which type of object is required. The user decides during runtime by clicking a button for example. If `type` contains the required type of object, you'll probably need to use an `if` or a `switch`, and do something like this:

```
var o;
if (type === 'Image') {
  o = new MYAPP.dom.Image();
}
if (type === 'Link') {
  o = new MYAPP.dom.Link();
}
if (type === 'Text') {
  o = new MYAPP.dom.Text();
}
o.url = 'http://...';
o.insert();
```

This works fine, but if you have a lot of constructors, the code might become too lengthy. Also, if you are creating a library or a framework, you might not even know the exact names of the constructor functions in advance. In such cases, it's useful to have a factory function that takes care of creating an object of the dynamically determined type.

Let's add a factory method to the `MYAPP.dom` utility:

```
MYAPP.dom.factory = function(type) {
  return new MYAPP.dom[type];
}
```

Now you can replace the three `ifs` with the simpler:

```
var o = MYAPP.dom.factory(type);
o.url = 'http://...';
o.insert();
```

The example `factory()` method above was simple, but in a real life scenario you'll probably want to do some validation against the `type` value and optionally do some setup work common to all object types.

Decorator

The *Decorator* design pattern is a *structural* pattern; it doesn't have much to do with how objects are created but rather how their functionality is extended. Instead of using inheritance where you extend in a linear way (parent-child-grandchild), you can have one base object and a pool of different decorator objects that provide extra functionality. Your program can pick and choose which decorators it wants and in which order. For a different program, you might have a different set of requirements and pick different decorators out of the same pool. Take a look at how the usage part of the decorator pattern could be implemented:

```
var obj = {
  function: doSomething() {
    console.log('sure, asap');
  },
  // ...
};
obj = obj.getDecorator('deco1');
obj = obj.getDecorator('deco13');
obj = obj.getDecorator('deco5');
obj.doSomething();
```

You can see how you can start with a simple object that has a `doSomething()` method. Then you can pick some of the decorator objects (identified by name) you have lying around. All decorators provide a `doSomething()` method which first calls the same method of the previous decorator and then proceeds with its own code. Every time you add a decorator, you overwrite the base `obj` with an improved version of it. At the end, when you are finished adding decorators, you call `doSomething()`. As a result all of the `doSomething()` methods of all the decorators are executed in sequence. Let's see an example.

Decorating a Christmas Tree

Let's illustrate the decorator pattern with an example: decorating a Christmas tree. You start with the `decorate()` method.

```
var tree = {};
tree.decorate = function() {
  alert('Make sure the tree won\'t fall');
};
```

Now let's implement a `getDecorator()` method which will be used to add extra decorators. The decorators will be implemented as constructor functions, and they'll all inherit from the base `tree` object.

```
tree.getDecorator = function(deco) {
    tree[deco].prototype = this;
    return new tree[deco];
};
```

Now let's create the first decorator, `RedBalls()`, as a property of `tree` (in order to keep the global namespace cleaner). The `RedBall` objects also provide a `decorate()` method, but they make sure they call their parent's `decorate()` first.

```
tree.RedBalls = function() {
    this.decorate = function() {
        this.RedBalls.prototype.decorate();
        alert('Put on some red balls');
    }
};
```

Similarly, adding a `BlueBalls()` and `Angel()` decorators:

```
tree.BlueBalls = function() {
    this.decorate = function() {
        this.BlueBalls.prototype.decorate();
        alert('Add blue balls');
    }
};
tree.Angel = function() {
    this.decorate = function() {
        this.Angel.prototype.decorate();
        alert('An angel on the top');
    }
};
```

Now let's add all of the decorators to the base object:

```
tree = tree.getDecorator('BlueBalls');
tree = tree.getDecorator('Angel');
tree = tree.getDecorator('RedBalls');
```

Finally, running the `decorate()` method:

```
tree.decorate();
```

This single call results in the following alerts (in this order):

- **Make sure the tree won't fall**
- **Add blue balls**
- **An angel on the top**
- **Put some red balls**

As you see, this functionality allows you to have as many decorators as you like, and to choose and combine them in any way you like.

Observer

The observer pattern (also known as the subscriber-publisher pattern) is a behavioral pattern, which means that it deals with how different objects interact and communicate with each other. When implementing the observer pattern you have the following objects:

- One or more publisher objects that announce when they do something important, and
- One or more subscribers that are tuned in to one or more publishers and listen to what the publishers announce, then act appropriately

The observer pattern may sound familiar to the browser events discussed in the previous chapter, and rightly so, because the browser events are one example application of this pattern. The browser is the publisher: it announces the fact that an event (such as `onclick`) has happened. Your event listener functions that are subscribed to (listen to) this type of event will be notified when the event happens. The browser-publisher sends an event object to all of the subscribers, but in your custom implementation you don't have to use event objects, you can send any type of data you find appropriate.

There are two subtypes of the observer pattern: push and pull. Push is where the publishers are responsible for notifying each subscriber, and pull is where the subscribers monitor for changes in a publisher's state.

Let's take a look at an example implementation of the push model. Let's keep the observer-related code into a separate object and then use this object as a mixin, adding its functionality to any other object that decides to be a publisher. In this way any object can become a publisher and any function object can become a subscriber. The observer object will have the following properties and methods:

- An array of subscribers that are just callback functions
- `addSubscriber()` and `removeSubscriber()` methods that add to and remove from the subscribers array
- A `publish()` method that takes data and calls all subscribers, passing the data to them
- A `make()` method that takes any object and turns it into a publisher by adding all of the above methods to it

Here's the observer mixin object which contains all the subscription-related methods and can be used to turn any object into a publisher.

```
var observer = {
  addSubscriber: function(callback) {
    this.subscribers[this.subscribers.length] = callback;
  },
  removeSubscriber: function(callback) {
    for (var i = 0; i < this.subscribers.length; i++) {
      if (this.subscribers[i] === callback) {
        delete(this.subscribers[i]);
      }
    }
  },
  publish: function(what) {
    for (var i = 0; i < this.subscribers.length; i++) {
      if (typeof this.subscribers[i] === 'function') {
        this.subscribers[i](what);
      }
    }
  },
  make: function(o) { // turns an object into a publisher
    for(var i in this) {
      o[i] = this[i];
      o.subscribers = [];
    }
  }
};
```

Now let's create some publishers. A publisher can be any object; its only duty is to call the `publish()` method whenever something important occurs. Here's a `blogger` object which calls `publish()` every time a new blog posting is ready.

```
var blogger = {
  writeBlogPost: function() {
    var content = 'Today is ' + new Date();
    this.publish(content);
  }
};
```

Another object could be the *LA Times* newspaper which calls `publish()` when a new newspaper issue is out.

```
var la_times = {
  newIssue: function() {
    var paper = 'Martians have landed on Earth!';
    this.publish(paper);
  }
};
```

Turning these simple objects into publishers is pretty easy:

```
observer.make(blogger);
observer.make(la_times);
```

Now let's have two simple objects `jack` and `jill`:

```
var jack = {
  read: function(what) {
    console.log('I just read that ' + what)
  }
};
var jill = {
  gossip: function(what) {
    console.log('You didn\'t hear it from me, but ' + what)
  }
};
```

`jack` and `jill` can subscribe to the `blogger` object by providing the callback methods they want to be called when something is published.

```
blogger.addSubscriber(jack.read);
blogger.addSubscriber(jill.gossip);
```

What happens now when the `blogger` writes a new post? The result is that `jack` and `jill` get notified:

```
>>> blogger.writeBlogPost();
```

**I just read that Today is Sun Apr 06 2008 00:43:54 GMT-0700
(Pacific Daylight Time)**

**You didn't hear it from me, but Today is Sun Apr 06 2008 00:43:54 GMT-0700
(Pacific Daylight Time)**

At any time, `jill` may decide to cancel her subscription. Then when writing another blog post, the unsubscribed object is no longer notified:

```
>>> blogger.removeSubscriber(jill.gossip);
```

```
>>> blogger.writeBlogPost();
```

**I just read that Today is Sun Apr 06 2008 00:44:37 GMT-0700
(Pacific Daylight Time)**

`jill` may decide to subscribe to *LA Times*, as an object can be a subscriber to many publishers.

```
>>> la_times.addSubscriber(jill.gossip);
```

Then when *LA Times* publishes a new issue, `jill` gets notified and `jill.gossip()` is executed.

```
>>> la_times.newIssue();
```

You didn't hear it from me, but Martians have landed on Earth!

Summary

In this final chapter, you learned about some common JavaScript coding patterns and learned how to make your programs cleaner, faster, and better at working with other programs and libraries. Then you saw a discussion and sample implementations of some of the design patterns from the *Book of Four*. You can see how JavaScript is a fully-featured dynamic object-oriented programming language and that implementing classical patterns in a dynamic language is pretty easy. The patterns are, in general, a large topic and you can join the author of this book in a further discussion of the JavaScript patterns at the web site JSPatterns.com.

You now have sufficient knowledge to be able to create scalable and reusable high-quality JavaScript applications and libraries using the concepts of Object-Oriented Programming. *Bon voyage!*